Implementing a Global Address Space Language on the Cray X1: the Berkeley UPC Experience

Christian Bell   Wei Chen
CS252 & CS262 Project
December 15, 2003

Abstract

The Berkeley UPC Compiler is an open source, high performance and portable implementation of Unified Parallel C (UPC), an SPMD global-address space language extension of ISO C. In previous work, we have experimented our compiler on a variety of high-performance networks and parallel architectures, including distributed memory machines and clusters of SMPs. Our goal in this paper is to implement and analyze the performance of the Berkeley UPC Compiler on the newly introduced Cray X1 system, a vector capable NUMA-based supercomputer that offers both low communication latency and high network bandwidth through a shared memory programming interface. We provide empirical performance characterizations of the communication primitives through micro-benchmarks, evaluate their effectiveness in compiling for a global address model and for UPC in particular, and implement the Berkeley UPC compiler based on these observations. Finally, we evaluate the performance of our implementation with a number of serial and parallel application kernel benchmarks.

1 Introduction

Global Address Space (GAS) languages have recently emerged as a promising alternative to the traditional message passing model for parallel applications. Designed as parallel extensions for popular sequential programming languages, languages such as UPC [14], Titanium [23], and Co-Array Fortran [20] provide better programmability through the support of a user-level global address space, leading to more flexible remote accesses with one-sided communications. GAS languages thus offer a more conventional programming style compared to the message passing model, and good performance can still be achieved since programmers retain explicit control of data placement and load balancing. Another virtue of GAS languages is their versatility; while it has not yet reached the level of MPI's ubiquity, UPC implementations are now available on a significant number of platforms, ranging from multiprocessors to the many flavors of network of workstations.

In the class of tightly integrated multiprocessors, the Cray X1 has recently been introduced as a system in the line of massively parallel processors, with the particular distinction of delivering powerful vector processing over non-uniform shared memory. The Cray X1 presents an interesting case study for GAS languages, which this document attempts to address through a complete implementation of the Berkeley UPC compiler. On top of simplifying communication operations as direct reads and writes to remote memory locations, the raw performance is impressive in terms of communication (peak memory bandwidth and low communication latency) as well as computation (powerful vector pipelines). Furthermore, efficient hardware support for strided accesses and scatter/gather operations has the potential of reducing substantial overheads associated to fine-grained remote accesses. Such an array of features would appear to be quite suitable for languages such as UPC that employ a distributed shared memory programming paradigm.

Our experiences, however, suggest that more efforts may still be required before the X1 can be considered an ideal architecture for UPC and GAS languages in general. One notable pitfall is the lack of split-phase read operations, which removes most opportunities in communication and computation overlapping. Additionally, the over-reliance on vectorization to achieve performance speedups can result in poor performance for a large class of applications that are not suitable for vectorization. The absence of a rich set of user-level communication primitives, in particular X1's lack of per-operation completion guarantees, also causes implementation difficulties and performance inefficiencies.

The rest of the paper is organized as the follows. Section 2 describes UPC, Berkeley UPC, and the Cray X1 system. Section 3 details our implementation of the GASNet communication layer, while Section 4 discusses our strategy for achieving good serial performance. Section 5 summarizes our optimizations of the shared pointer operations, followed by Section 6's discussion of the communication optimization opportunities on the Cray X1. Section 7 evaluates our compiler's parallel performance, and finally Section 8 concludes the paper with an evaluation of the Cray X1 architecture.

2 Background

2.1 Unified Parallel C

UPC (Unified Parallel C) is a parallel extension of the C programming language aimed at supporting high performance scientific applications. The language adopts the SPMD programming model, so that every thread runs the same program but

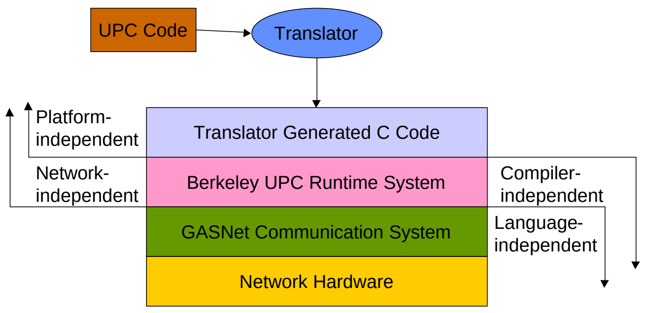


Figure 1. Architecture of the Berkeley UPC Compiler

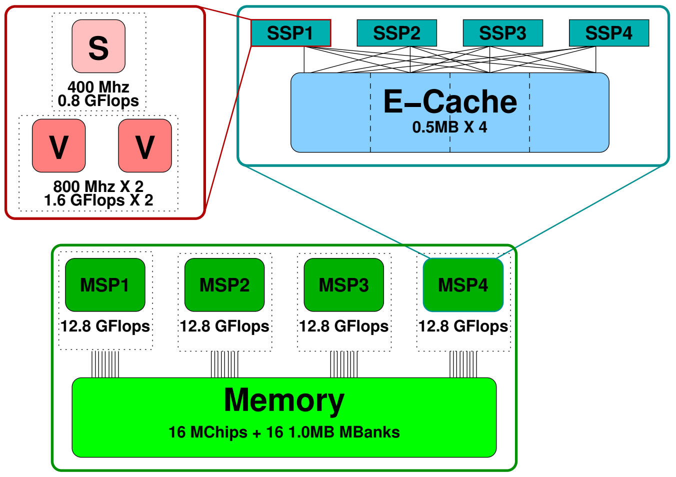


Figure 2. Cray X1 single node: Each MSP contains 4 SSPs each with 2 Vector and 1 Scalar Unit

keeps its own private local data. In addition to each thread's private address space, UPC provides a shared memory area to facilitate communication among threads, and programmers can declare a shared object by specifying the `shared` type qualifier. While a private object may only be accessed by its owner thread, all threads can read or write to data in the shared address space. Because the shared memory space is logically divided among all threads, from a thread's perspective the shared space can be further divided into a local shared memory and remote one. Data located in a thread's local shared space are said to have "affinity" with the thread, and compilers can utilize affinity information to exploit data locality in applications to reduce communication overhead.

UPC gives the user direct control over data placement through local memory allocation and distributed arrays. When declaring a shared array, programmers can specify a block size in addition to the dimension and element type, and the system uses this value to distribute the array elements block by block in a round-robin fashion over all threads. For example, a declaration of `shared [2] int ar[10]` tells the compiler to allocate the first two elements of `ar` on thread 0, the next two on thread 1, and so on. If the block size is omitted the value defaults to one (cyclic layout), while a layout of `[]` or `[0]` indicates indefinite block size, i.e., that the entire array should be allocated on a single thread. A pointer-to-shared thus needs three logical fields to fully represent the address of a shared object: `address`, `thread_id`, and `phase`. The `thread_id` indicates the thread that the object has affinity to, the `address` field stores the object's "local" address on the thread, while the `phase` field gives the offset of the object within its block. Other notable UPC features include a `upc_forall` parallel loop, dynamic allocation functions, synchronization constructs, and a choice between a strict or relaxed memory consistency model; consult the UPC language specification for more details [14].

2.2 The Berkeley UPC Compiler

Figure 1 shows the overall structure of the Berkeley UPC compiler [1], which is divided into three main components: the UPC-to-C translator, the UPC runtime system, and the GASNet communication system [5]. During the first phase of compilation, the Berkeley UPC compiler preprocesses and translates UPC programs into ANSI-compliant C code in a platform-independent manner, with UPC-related parallel features converted into calls to the runtime library. The translated C code is then compiled using the target system's C compiler and linked to the runtime system, which performs initialization tasks such as thread generation and shared data allocation. The Berkeley UPC runtime delegates communication operations to the GASNet communication layer, which provides a uniform interface for low-level communication primitives on all networks.

We believe this three-layer design has several advantages. First, because of the choice of C as our intermediate representation, our compiler will be available on most hardware platforms that have an ANSI-compliant C compiler; the currently available UPC Compilers only support specific systems. Second, both the UPC runtime system and GASNet implement a well-defined interface: the runtime offers a flexible shared pointer abstraction with the option of running multiple threads per node and GASNet implements network-independent Global-Address Space primitives. This two-tier approach can be tailored to move more or less functionality into the runtime or GASNet based on how close either layer can target native communication primitives. In a previous work [8], we have validated our design by showing that, in spite of the modularity used to support portability, the Berkeley UPC Compiler performs well in both absolute and relative terms. In an absolute sense, the communication performance is very close to that of the underlying network hardware; in a relative sense, the compiler is competitive with another UPC compiler.

2.3 Cray X1

The Cray X1 [11] is a supercomputer system developed by Cray which combines powerful vector processors with high bandwidth network interconnects through a shared memory programming model. Figure 2 illustrates the architecture of a Cray X1 node, the building blocks of the system. Each node consists of four multi-streaming processors (MSP), with a globally shared physical memory and 2MB individual caches. Each MSP in turn is composed of four single-streaming processors (SSP), formed with two vector pipelines and one scalar processor. The architecture offers a global address space, so that a processor can directly address memory locations on another node. From an application's point of view, the Cray X1 system thus behaves as a Non-Uniform Memory Access (NUMA) architecture, with the main distinction being that inter-node accesses are not cacheable. The Cray *shmem* API [18] provides an uniform interface for both local and remote memory accesses, as well as providing various atomic and synchronization operations.

Cray X1 offers two configurations for executing parallel programs. Explicit parallelism is achieved in the SSP mode by treating each SSP as a separate processor, permitting up to 16 CPUs per node. The alternative MSP mode maps an execution thread to an MSP, and utilizes compiler-directed *multi-streaming* to accomplish automatic parallelization. The multistreaming process divides either vectorized inner loop or unvectorized outer loop into four independent segments, and assigns them to different SSPs to be executed in parallel. The theoretical peak performance of the different processing units is also included in Figure 2. An early performance evaluation of the Cray X1 [13] suggests that many parallel applications can achieve significant performance with enough porting and optimization efforts.

Cray X1 also supports a variety of major parallel programming models, including a UPC compiler that implements a subset of the UPC specification. Important missing features include block cyclic pointers, `upc_forall` loops, non-collective shared memory allocation, and restrictions on the block size of shared arrays. While some of the deferred features (e.g., forall loops) merely offer syntactical convenience and optimization hints, many provide essential functionalities of UPC applications and have no easy workarounds without affecting program behavior. Their exclusion for performance or implementation complexity reasons thus severely limits the usefulness of the compiler, which in our experiments fails to compile several NAS UPC benchmarks. In contrast, our Berkeley UPC implementation is fully UPC 1.1 compliant, and in later sections we will discuss the performance tradeoffs involved with some of the features.

The Cray X1 differs considerably in its support for remote memory accesses from its predecessors. The Cray T3D provided three different remote memory access mechanisms: direct loads and stores, a 16 entry memory word prefetch queue as well as a block transfer engine for bulk asynchronous transfers [3]. Later, the Cray T3E recognized the overhead and difficulties in maintaining three access methods and suggested E-registers as an efficient user-level mechanism for providing split-phase remote memory operations [21]. Although E-registers must be explicitly managed by the user, they are general enough to implement direct and strided remote memory accesses, message queues for message-passing communication as well as an array of versatile synchronization primitives (virtual network barriers and eureka synchronizations). Of these mechanisms, the X1 has only retained the ability to execute transparent global loads and stores, effectively removing any user-level messaging ability Gets, translated either into scalar or vector loads, are entirely blocking primitives, leaving no opportunity for overlapping communication with computation. Puts are translated into stores which, not unlike write buffers in out-of-order processors, allow some computation to run ahead of communication. However, the X1 has retained the *memory centrifuge* property of the T3E to easily manipulate global pointers – memory allocated on the symmetric heap and static application data differs only in the Process Element (PE) portion of the global virtual memory address.

3 The X1 GASNet Communication Layer

It is generally believed that the Message-Passing Interface is one of the only mechanisms available to guarantee portability of high performance computing codes. Although it would seem that MPI is the right way to guarantee total system portability, the case for performance cannot be easily made in the context of a compilation target. It has recently been demonstrated in [6] that both the 1.1 and the newly revised 2.0 MPI specification provide restrictive remote memory access semantics that prevent its use as a compilation target. The authors also show that the performance of the network interconnect and underlying low-level networking software is best leveraged by exposing less restrictive and more expressive semantics, such as those proposed by GASNet.

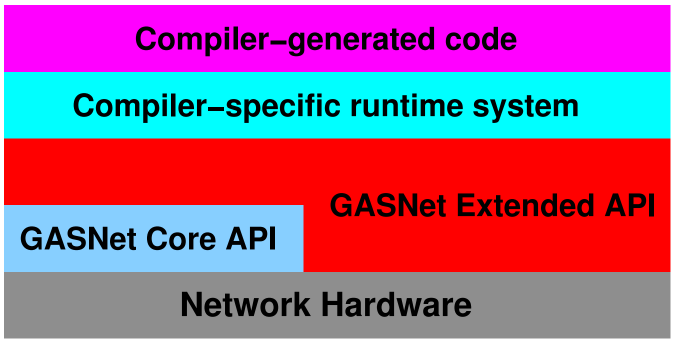


Figure 3. GASNet communication system: some functionality in the general core API can be bypassed by network-specific primitives in the Extended API

The approach to porting GASNet to a new platform entails starting from a provided template conduit and proceeding in a two phase implementation. First, developers target the GASNet Core API, based on Active Messages [17], after which a reference version of the GASNet Extended API can be hooked in to provide a complete GASNet implementation. Second, primitives available in the reference extended API can be swapped in for more efficient network primitives offered by the underlying networking software. Based on prior experience porting GASNet to five other networks, we have found this approach to be very effective in quickly obtaining a working conduit and gradually refining it with more efficient primitives. After sufficiently experimenting with the X1's communication software, we were able to develop a working conduit within a matter of days. The remainder of this section provides details about the X1's network architecture, exposes some of the issues in providing a shared memory implementation of Active Messages, and provides details about the GASNet extended API.

3.1 X1 Communication and Synchronization

The X1 constitutes an important departure for Cray from a message-oriented network to a purely shared memory platform. Although this hints at possibly tighter node integration, the net effect is an important reduction in the amount and flexibility of synchronization primitives for inter-node communication. The platform provides minimal memory ordering guarantees, and explicit instructions are necessary to maintain scalar/vector and vector/vector interactions [9]. The most important forms of these instructions, available as compiler intrinsics, are summarized in table 1. Vector instructions can only be generated when high-level recurrence-free C loops are recognized as vectorizable by the Cray C compiler. As a result, only `gsync` and `msync` can make assumptions about the generated code as they don't depend on any particular scalar of vector instruction being generated.

Instruction	*Description*
`gsync`	Global ordering of all prior references
`msync`	MSP ordering of all prior references
`lsync S,V`	Local ordering of all prior scalar refs before later vector refs
`lsync V,S`	Local ordering of all prior vector refs before later scalar refs
`lsync V,V`	Local ordering of all prior vector refs before later vector refs

Table 1. X1 Memory Ordering Instructions

Although reducing synchronization and ordering semantics to a single global ordering instruction simplifies the view of the system, platforms that offer per-operation synchronization permit both communication and computation to be scheduled in split (often overlapping) phases. Such was the case on the Cray T3E where communication could be scheduled asynchronously by programming user-level network registers (E-registers) to access remote memory. The slower 400Mhz scalar unit on the X1 is one possible explanation for overlooking E-registers, since it may now run behind the network link bandwidth. This wasn't the case on the T3E – the processor could remain ahead of the link even if issuing a get operation required two off-chip memory accesses [21]. A second explanation resides in the presence of a vector unit that can, by pipelining loads and stores, issue both both strided and indexed strided gets and puts directly through transparent global pointers. In fact, by moving to a transparent global memory access mechanism, the burden of efficiently translating global memory references and managing E-registers has been lifted, which simplifies the task of a compiler. Relying on a model that also integrates global communication with vector computation, however, poses interesting questions for GASNet.

3.2 Implementing the Core API: Shared Memory Active Messages

The GASNet porting effort starts with an implementation of the Core API, which on the X1 essentially amounts to enabling active messages to be sent and received through shared memory. Since most of the emphasis of the Core is to simply provide a working implementation, we have targeted the *shmem* layer for memory transfers.

Work on implementing shared memory AM has previously been explored in the context of clusters of SMPs [17]. The authors then acknowledged the need for minimizing synchronization overhead in the presence of high contention for a shared queue and proposed a lock-free algorithm for reserving slots in the queue. Their lock-free algorithm works in two phases: the first uses a compare and swap to optimistically increment the current slot in the queue and the second uses a compare and swap to busy wait until the slot is free. While this approach may be viable for NUMA or SMP bus-based architectures that cache memory locations, the X1 does not cache remote locations, which prevents it to busy wait on an in-cache memory line to be subsequently invalidated. We've kept the busy-wait phase of the algorithm but substituted the first compare and swap that selects a slot in the queue since it mainly serves the purpose of distributing requests over all possible queue slots. For the X1, the fetch&increment atomic primitive provided the best performance overall.

The largest performance improvement on the X1 was obtained by using an atomic compare&mask operation to post completions for individual queue slots. Instead of posting a `DONE` state directly in the busy wait queue, we use an array of 64-bit bit-fields and set them using compare and mask operations. Incidentally, since GASNet is polling-based and hence sensitive to the cost of `AMPoll()`, we have also addressed the problem of keeping polling overhead low – for queue sizes of less than 64 slots, the overhead is limited to a single word load.

3.3 Implementing the Extended API

On all recent Cray platforms, the *shmem* interface is presented as the ideal interface to target in order to exploit full network potential. For example, MPI is an ideal client for the *shmem* interface since it is based on a bulk synchronous programming approach, where high-bandwidth bulk communication phases are interspersed with local computation. While we have found *shmem* useful with regards to large-message performance and quick prototyping, it lacks expressiveness in synchronization mechanisms and presents an additional source of overhead for small messages. Therefore, we have left the bandwidth limited operations to shmem and have optimized many of the latency-sensitive GASNet primitives to directly manipulate global address pointers.

The X1 allows programmers to take advantage of a symmetric heap through the hardware *memory centrifuge* for dynamically allocated memory, which affects the global address

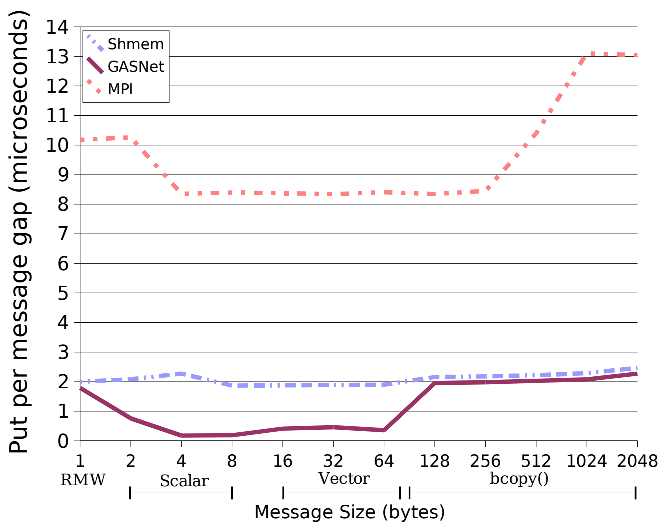


Figure 4. Small Put Performance

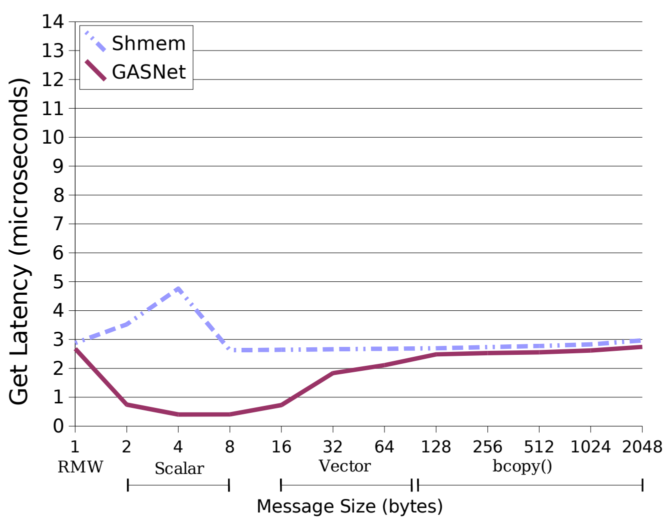


Figure 5. Small Get Performance

pointer representation and ultimately facilitates inter-node communication. By using the symmetric heap allocator to allocate an equal amount of memory across all nodes in the parallel job, the programmer is guaranteed that only the process element (PE) high bits portion of the address will change in the pointer representation. If a globally-shared segment is allocated at startup and each node's segment base address known, a translation macro is sufficient to prefix to any GASNet communication primitive to allow inter-node communication through global pointers.

While Cray provides the `bcopy()` call as a mechanism for bulk data transfers, scalar and vector loads and stores through global pointers provide the best alternative for smaller data sizes. For the non-bulk put/get GASNet variants, we established 10 words (80 bytes) as a threshold between issuing direct loads/stores and turning over the data transfer to `bcopy()`. Figures 4 and 5 show how GASNet compares to both *shmem* and MPI in terms of small message performance for one-sided non-blocking put and get operations. Since put operations are essentially store operations, the put figure measures the per message gap, or the amount of time the processor is tied up when repeatedly injecting messages into the network. As a frame of reference for comparing GASNet to the Message-passing Interface, MPI numbers are included for measuring the gap in issuing put operations. Conversely, the get figure measures the total load latency as the load operations are indivisible. For small sizes close to the 8-byte word size, translating the put to global store operations is clearly beneficial (1-byte put operations require a read-modify-write operation, which explains their poor performance).

A common source of overhead in multi-layer software architectures is the overhead caused by the multiple levels of translation. For shared-memory platforms, which can access pointers using loads and stores, the actual remote load is effectively translated after being translated from a pointer-to-shared and subsequently passed to GASNet. For loosely-coupled platforms, this is less of a problem since a single word puts/gets already require library calls to the underlying network software to initiate and synchronize network each operation. It would seem that this problem is further aggravated on the X1, where the compiler is ultimately responsible for identifying and optimizing both communication and computation for vectorization. For example, if a 1 word put primitive is inserted in a tight loop, vectorization is shut down if the put is carried out through a function call. Fortunately, the Berkeley UPC compiler, by way of the runtime and GASNet, supports full inlining of shared pointer manipulation and global communication, thus allowing all traces of functions calls to be removed from performance-critical UPC code. As such, the X1 GASNet extended API is beneficial for the purpose of exposing vectorization on common vector-sized elements and provides large-message bandwidth similar to *shmem*.

4 Serial Performance

As mentioned in Section 2.2, the Berkeley UPC Compiler performs high-level transformations on UPC programs and generates C code as its intermediate output. This modular design greatly simplifies the efforts for supporting new architectures; no changes are required for the translator, whose code generation already accommodates for different architectural parameters such as register size and the integral type width. While our design brings considerable flexibility, it also has potential undesirable implications on the sequential performance of UPC programs. As UPC itself is an extension of ISO C, the majority of an UPC application will still be written as ordinary C code. Although the translator should preserve the semantics of the sequential part of the program, it is infeasible to expect the translated output to be syntactically identical to the program source, due to optimizations performed by the translator and the non one-to-one mapping between its intermediate representation and the C language. Ensuring the performance of the translated code to be comparable to that of the original sequential code thus presents a formidable challenge, as one must account for the idiosyncrasies of the various backend C compilers. This is especially critical for achieving good performance on the Cray X1, where the vector units can execute orders of magnitude faster than it sequential counterpart, and slight modifications to an inner loop such as the presence of a redundant cast could inhibit its vectorization.

Our goal is to evaluate the serial performance of the Berkeley UPC Compiler, concentrating on its ability to maintain the vectorizability of the sequential portion of the program. With full optimizations enabled, the Cray C compiler [10] performs automatic vectorization on expressions inside a loop that it detects to be free of cycles of dependences, after applying vectorization-enabling transformations such as inlining, loop splitting, and loop interchange. The compiler also vectorizes certain special recurrences such as reduction and scatter/gather. Cray C provides two program level optimization techniques to assist the compiler's alias and dependence analysis in identifying candidates for vectorization: `restrict` pointers and the `ivdep` pragma. The former is equivalent to the restrict type qualifier in the ISO C99 standard, and essentially declares the pointer to be free of aliases in the current scope. The `ivdep` pragma asserts that no vector dependences exist within the loop immediately following the pragma. Another extremely useful directive, `concurrent`, indicates no recurrences exist between array accesses. As such, our translation strategy is simply to keep the translated output as syntactically similar as possible to the original source. The level of the intermediate representation is kept sufficiently high such that C loops are preserved in its original form (with `for` loops converted into the equivalent `while` loop). Similarly, array expressions are recognized and handled specially by the translator, both to allow for more aggressive transformations by its optimizer and to provide the C compiler with more precise information; an array access $ar[exp]$ is translated into the identical $*(ar + exp)$, still a vectorizable expression for Cray C. Multidimensional arrays have their index expressions linearized to behave like one dimensional arrays, and thus requires an additional cast to the appropriate pointer type on the array variable. As the Berkeley UPC Compiler is compliant to the C99 standard, it already supports restrict-qualified pointers, and source level pragmas used by the backend compiler are uninterpreted by the translator and appear in the same location in the output.

4.1 Livermore Kernels

We chose the C version of the Livermore Kernels [19] to conduct a thorough evaluation of the serial performance of our compiler. The Livermore Loops consist of 24 computation loops extracted from common scientific applications, and therefore should closely reflect the computation performance offered by our compiler. In particular, X1's reliance on the vector unit to achieve both fast computation and high memory bandwidth means that application performance will often hinge on whether its main computation loops could be efficiently vectorized. Table 2 presents the aggregate performance for both the original C source and the translated output with the `-O3` flag, while Figure 6 displays each kernel's individual performance.

The aggregate data seem to imply that the Berkeley UPC Compiler's translation process incurs a substantial overhead; average rate is decreased by nearly one half, while the geometric mean, a more accurate indicator of performance, also goes down by 30%. A closer examination of the individual benchmarks, however, reveals that most of the performance degradation can be attributed to a small subset of the kernels. Kernel 8 results in the largest performance gap, and is the only benchmark where our compiler could not preserve the vectorizability of the original loop. The anomaly can be attributed the linearizing of several three dimensional array accesses in the translated code, which confuses the Cray C compiler into identifying non-existing recurrences between them. We are investigating the issue by preserving multidimensional array accesses in its original form. Another notable difference comes from Kernel 21, which performs a naive matrix multiplication; although the translated code is also vectorizable, Cray C is able to recognize this special pattern in the original code and apply loop interchange and unrolling to further enhance its performance. Since the translated output exhibits similar performance to the C code for most of the kernels, we thus expect the Berkeley UPC compiler to offer competitive serial performance on a vector platform like the Cray X1. Finally, we note that Cray C has failed to vectorize a substantial number of the benchmarks, even though many of them do not contain any vector dependences. This suggests that automatic vectorization alone is not sufficient for good computation performance due to limitations of static analysis, and it is therefore important for the Berkeley UPC compiler to preserve the programmer-supplied vectorization-enhancing pragmas in the translated code.

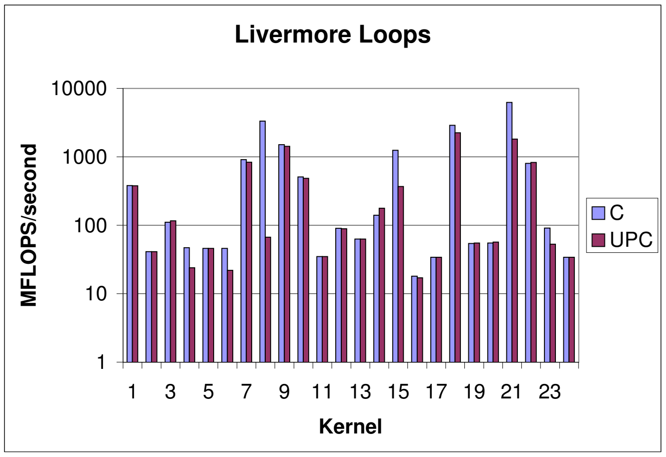


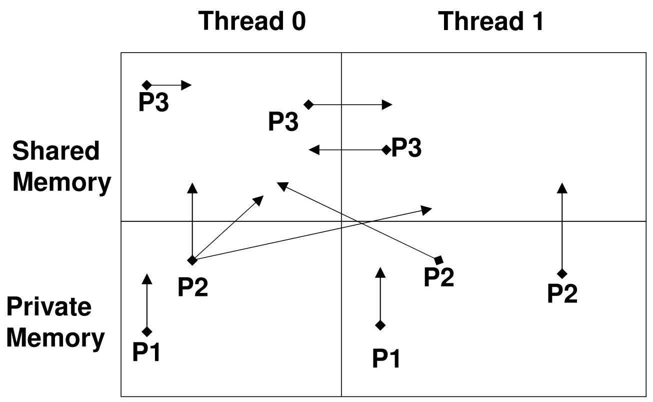
Figure 6. Performance of the Individual Livermore Kernels

	Geo. Mean	Avg. Rate	Har. Mean	Max	Min
C	160	738	59.4	6232	8.9
UPC	115	408	47.6	4495	5.1

Table 2. Aggregate Performance of the Livermore Loops (in MFLOPS/second)

5 Implementing Pointer-to-shared Operations

Figure 7 illustrates three different kinds of UPC pointers: private pointers pointing to objects in the thread's own private space (P1 in the figure), private pointers pointing to the shared address space (P2), and pointers living in shared space that also point to shared objects (P3). Compared to a regular C pointer, a generic pointer-to-shared contains two additional `id` and `phase` fields. Figure 8 demonstrates how the three components can be used in combination to encode a shared value. When performing pointer arithmetic on a pointer-to-shared, one therefore may need to update all three fields, making the operation inevitably slower than private pointer arithmetic. To overcome this overhead, the Berkeley UPC Compiler implements an optimization called "phaseless" pointers for the common special case of cyclic and indefinite pointers. Cyclic pointers have a block size of one, and their phase is thus always zero;



P1 == private pointer
P2 == pointer-to-shared
P3 == shared pointer-to-shared

Figure 7. Types of Pointers in UPC.

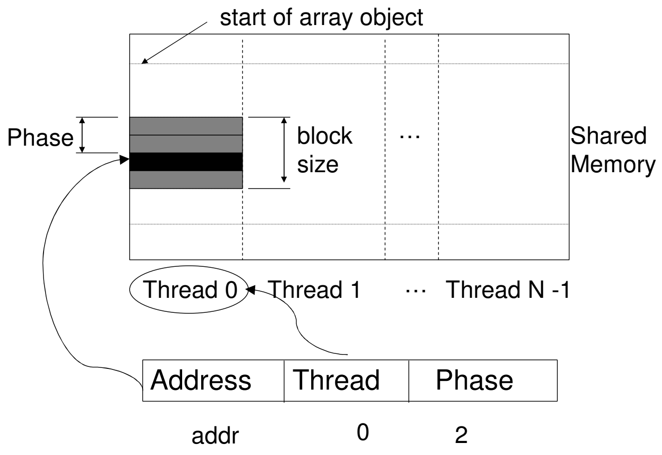


Figure 8. Pointer-to-shared components in UPC.

Indefinite pointers have a block size of zero, and their phase is also defined to zero since all elements belong to the same UPC thread. Cyclic and indefinite pointers are thus "phaseless", and our compiler exploits this knowledge to schedule more efficient operations for them. Experimental results [8] show that the optimization is effective in improving the performance of shared pointer arithmetic, by shredding 50% of the overhead off cyclic pointers and making indefinite pointers almost as fast as C pointers for pointer-integer addition.

5.1 Cray X1 Specific Optimizations

Given the success of our phaseless pointer optimization, we naturally want to exploit the global address space offered by the Cray X1 *memory centrifuge* by exploring alternative pointer-to-shared representations. The first step is to ensure that the pointer-to-shared representation deviates from Cray X1's global pointers as little as possible. To implement its global address space abstraction across nodes with distributed memory, the Cray X1's pointer format closely resembles the structure of our phaseless pointers, as the global virtual address is composed of a process element number [1] and a local address field. We therefore represent phaseless pointers-to-shared directly as regular C pointers, eliminating the overhead associated with an additional level of indirection. Generic pointers-to-shared present more obstacles, as UPC semantics require that phase information can be extracted from arbitrary pointer-to-shared to permit easy indexing into the beginning of a block. The phase field is thus an intrinsic part of pointers to block-cyclically distributed shared data, and must be explicitly stored in the pointer construct. Through simple bit-level operations, the UPC pointer-to-shared representation can be mapped to meet the Cray specification for global address pointers. The representations for phased and phaseless pointers-to-shared as well as Cray global pointers are shown in figure 9.

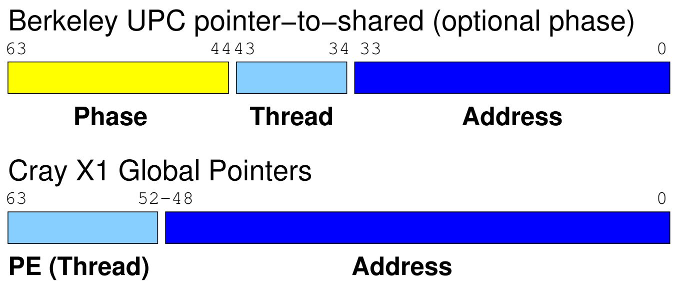


Figure 9. Cray Global-Address Space and Berkeley UPC pointer representations

In Section 3, we observed that GASNet's implementation of remote accesses adds little overhead over the assembly load and store instructions. Efficient scalar gets and puts, however, do not guarantee good communication performance on the X1, whose heavy reliance on vectorization for obtaining performance speedup presents interesting challenges for Berkeley UPC. Specifically, vectorizing fine-grained remote accesses within computation loops can dramatically improve the performance of programs written in shared memory style. Our micro-benchmarking of a simple vector addition loop shows that the vectorized unit stride accesses runs nearly two orders of magnitude faster than its scalar counterpart, while hardware scatter/gather instructions also outperforms scalar accesses by at least a factor of ten. Vectorization is thus crucial not only for serial performance but also for communication operations of fine-grained programs; while other platforms can make use of GASNet's rich set of non-blocking primitives to overlap communication with either more communication or computation,

[1]either a MSP or a SSP depending on compilation options

the X1 can only harness similar performance characteristics by vectorizing memory accesses.

Consequently, we carefully tuned the shared memory access primitives to exclude constructs that could interfere with vectorization. All function calls are either inlined or replaced as macros, with the common cases of 4/8 byte transfers translated directly into load and store instructions. Taking advantage of Cray X1's global address space, we also eliminate branches in gets and puts that check a pointer-to-shared's affinity, since both local and remote accesses can be serviced in the same fashion. It should be stressed, however, that all the optimizations added to accommodate the X1 did not require any modifications to the GASNet and UPC runtime interfaces. In this sense, the X1 has been useful in validating our assumption that our highly optimized layered approach provides complete and efficient global-address space functionality for both tightly coupled systems and decoupled systems such as networks of workstations.

An interesting challenge for vectorizing shared memory accesses arises in implementing blocking put operations. GASNet's semantics of a blocking put require that the value being stored be completely written to the destination address prior to returning; Cray X1, however, does not provide hardware support that polls for the completion of remote writes, and the only alternative for mimicking this behavior is to issue a global memory barrier that enforces global ordering of all prior references before all later references. Not only is the global barrier overkill when all that is needed is the guarantee of the completion of the immediate access, but the global synchronization performed by the barrier effectively inhibits all compiler vectorization. Although this problem may be somewhat mitigated by generating split-phase puts and attempting to hoist the syncs out of loops, the presence of even a single such instruction renders a loop unvectorizable. Our solution, instead, is to take advantage of UPC's relaxed consistency model to eliminate altogether the barrier for relaxed writes. UPC supports both a strict and a relaxed memory model, and relaxed accesses can be freely reordered as long as local data dependencies are still preserved. Since Cray X1 maintains the program order for two scalar references to the same location, correct local data dependencies will be established, and there is therefore no need for explicit instructions to guarantee the completion of a put operation. This allows the Cray C compiler to freely vectorize scalar memory references and schedule synchronizations as necessary. While strict accesses require stronger ordering guarantees and thus do not benefit from this optimization, they occur with much lower frequencies and are much less performance-critical.

5.2 Results

5.2.1 Micro-benchmarking

We evaluate the effectiveness of our optimizations by comparing our communication performance with that of the Cray UPC Compiler. Figure 10 presents the execution time of the pointer-to-shared manipulation functions, while Figure 11 presents the respective memory access time. The benchmark is constructed to avoid vectorization and unrolling of the operations, so as to reflect their true costs. As the results show, the Berkeley UPC Compiler offers competitive performance on shared pointer arithmetic; block cyclic (generic) pointers, in particular, generates comparable overhead compared to that of cyclic pointers, indicating there should be little incentive for Cray UPC not supporting it performance-wise. The execution time of UPC shared remote accesses are very close to GASNet's get/put latencies, signifying the low overhead incurred by the runtime layer. A substantial difference in performance is also observed between blocking and non-blocking remote puts, which can be attributed to the cost of the global memory barrier.

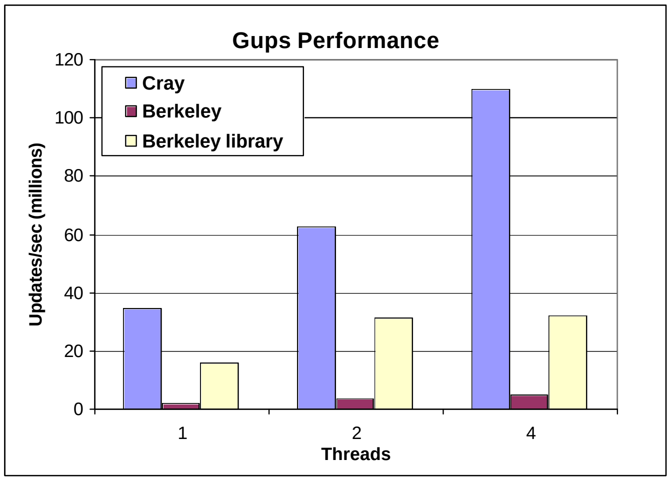


Figure 12. Gups Performance.

5.2.2 Gups

We next evaluate our compiler's ability in enabling vectorization of fine-grained communications by testing it with the $a[ind[i]]$ code pattern, which occurs frequently in scientific applications. The *gups* benchmark measures the compiler's performance on randomized remote accesses, as the program performs a series of read-modify-writes on random locations of a cyclically distributed shared array. Due to its native scatter/gather support, Cray X1 is likely the ideal architecture for this type of application.

```
shared long table[TABLE_SIZE];

/* Original UPC code */
for (i=0; i<64; i++)
    table[ idx[i] ] = t1[i];

/* Berkeley UPC translated code (no optimizations) */
for (i = 0; i < iters; i++) {
    temp = UPCR_ADD_PSHARED(table, 8, *(idx + i));
    UPCR_PUT_NB_PSHARED_VAL(temp, *(t1 + i));
}

/* Berkeley UPC with Gather/Scatter Memcpy */
upc_memcpy_scatter(table, t1, idx, 64);
```

Figure 13. Vectorization of high-level UPC scatter operation

Figure 12 shows the performance of the gups benchmark. Here Cray UPC substantially outperforms the unoptimized version of Berkeley UPC, due to Cray C's inability to vectorize

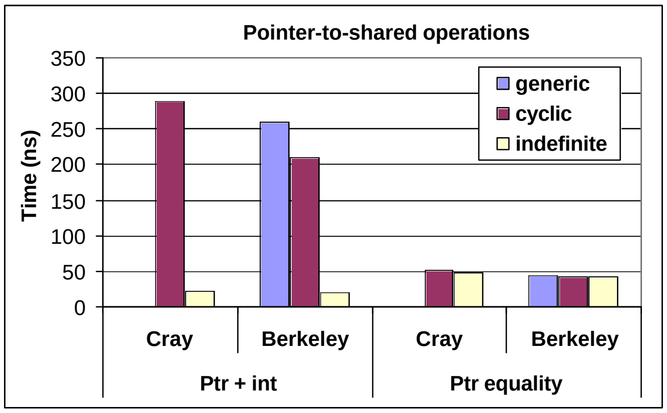


Figure 10. Performance of Shared Pointer Arithmetic. Results for generic pointers is missing for Cray UPC, since it does not support block cyclic pointers.

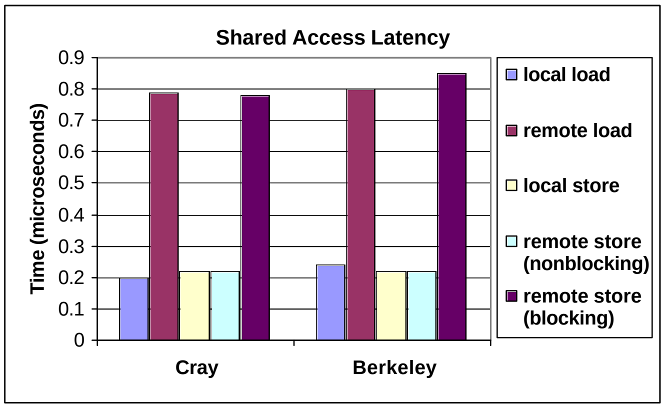


Figure 11. Execution Time of UPC Shared Memory Access

our translated code, shown in Figure 13. Because Cray UPC is tightly integrated with the C compiler, it could recognize the special pattern and efficiently schedule a hardware scatter-gather operation; the translated code generated by Berkeley UPC, on the other hand, could not be reliably converted into vector instructions, even when all function calls are inlined and the loop is clearly free of vector dependences. While implementing such loops directly with X1's vector assembly instructions may guarantee optimal performance, this presents a departure from our layered design and thus adversely affects the compiler's portability. Our solution is to provide instead special UPC scatter/gather library calls in the runtime layer. The functions can be targeted either at user-level or by the translator, and, as Figure 12 shows, can be implemented efficiently since all communications can be reduced to vectorizable instructions within an internal function. With a minimal change to the runtime API, this approach removes the difficulties in relying on the backend C compiler for recognizing special code patterns, and hence allows the code to be more portable with regards to cross-platform performance. Encouraged by the results, we are currently developing a UPC communication library that will include scatter/gather and other common communication idioms.

6 Potentials of UPC Compiler Optimizations

While working on the first official release of the Berkeley UPC Compiler, our efforts have concentrated on implementing the language features correctly, and as a result there still exists room for significant performance growth for UPC-specific compiler optimizations. In an earlier paper [8], we have identified several optimizations that prove to be valuable in a distributed memory environment: communication and computation overlap, prefetching of remote data, message aggregation, and privatization of local shared data. The performance characteristics that we have observed so far, however, raise questions about the appropriateness of these optimizations for the Cray X1. In this section, we evaluate the effectiveness of two important optimization techniques on the Cray X1.

6.1 Message Coalescing and Aggregation

The widely used LogGP network performance model [2] speaks volumes about the effectiveness of message coalescing and aggregation; by combining small puts and gets into large messages, not only does one save on the per-message startup overhead, but can also exploit the higher bandwidth offered by modern high-performance networks for large messages. Compiler support for this optimization is also crucial for UPC, whose use of global pointers tends to encourage a shared memory programming style that results in lots of small message traffic. The most common realization of this optimization, called *message vectorization*, significantly improves the performance of a fine-grained loop by copying all remote values it needs in one bulk transfer instead of issuing a read operation in every iteration. Other similar techniques include copying the entire struct when accessing its fields, and packing messages bound for the same destination node.

Our benchmarking of the Cray X1's memory and communication performance, however, raises doubt about the relevance of converting fine-grained accesses into coarse-grained bulk transfers on this platform. If the latencies and bandwidth of a remote memory access are comparable to those of the local access, it may not make sense to bulk fetch remote data into local buffers, since one still has to pay for the overhead of moving data from the main memory into cache. Furthermore, as we have seen in Section 5, hardware support for vectorized loads can alleviate much of the communication overhead for small messages. On the other hand, since memory across nodes is not kept cache coherent, if a remote shared object is to be referenced multiple times, it might be beneficial to copy the object locally so that its value resides in cache, as permitted by UPC's relaxed consistency model. Essentially, we seek to evaluate the impact of a shared memory programming paradigm for UPC application performance on Cray X1; if X1's transparent global loads and stores can efficiently support fine-grained accesses to non-local data, programmers can enjoy both the simplicity offered by a shared memory programming style and performance comparable to coarse-grained bulk communication.

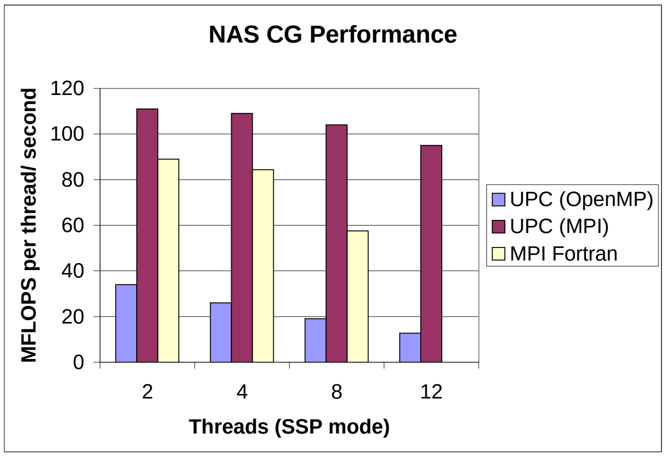


Figure 14. NAS CG (Class B): fine-grained vs. coarse-grained.

To answer this question, we compared the performance of two versions of the NAS conjugate gradient (CG) benchmark from [4]. The first is derived from an OpenMP shared memory implementation, with the exception that the column vector is replicated to avoid repeated random indexing into it. The second version is written in the style of one-sided coarse-grained communication, through the use of `upc_memget` library calls. The sparse matrix-vector multiplication in both versions was carefully tuned to ensure that the inner loops were vectorized. Both are compiled in SSP mode and executed such that the UPC threads are evenly distributed among the nodes. Performance results from the MPI Fortran version of the benchmark were also included for comparison[2]. As Figure 14 shows, performance of the shared memory version lags behind that of code with coarse-grained parallelism. The performance gap would also likely be much higher if the column vector was not replicated and instead accessed directly through pointers-to-shared. Much of the performance advantage offered by the coarse-grained version can be attributed to a tighter inner loop for the matrix-vector product, as the boundary information for each thread can be precomputed due to explicit partitioning of the sparse matrix. In summary, although Cray X1's tightly-coupled shared memory interface lowers the communication overhead, a coarse-grained communication model likely will still outperform a shared memory model even for applications with irregular and dynamic parallelism. This also suggests that UPC's hybrid programming model can be well-suited for Cray X1; fine-grained accesses through pointers-to-shared can deliver acceptable performance if they can be vectorized, while performance critical sections of the code can be further optimized into bulk synchronous transfers.

6.2 Communication/Computation Overlap

Compiler-controlled overlapping of communication and computation is a crucial optimization for parallel programs, as it can effectively hide communication overhead by keeping the processor busy with independent local computation while waiting for remote data to arrive. This capability is especially relevant for UPC programs; unlike other parallel programming paradigms such as MPI or split-C [12], which provide constructs such as `isend()/irecv()` and signaling stores, UPC currently offers no non-blocking communication operations at the language level and instead expects UPC compilers to perform such optimizations automatically. The straightforward implementation converts an one-sided blocking get/put operation into an initiation call and a corresponding synchronization call, with the compiler separating the two as far apart as possible and inserts independent computation or communication code in between. Several studies [24, 15, 7] have proposed global communication scheduling techniques that attempt to identify an optimal arrangement for all non-blocking memory accesses. Other variants of this optimization such as message strip mining [22] and software prefetching [16] are also useful in reducing an application's communication latencies.

The applicability of communication and computation overlap, however, is severely limited on the Cray X1. As Section 3 mentions, Cray X1 offers only a load and store based interface for remote communications, and the fact that loads are blocking renders useless any optimization that overlaps communication and computation through split phase accesses. While Cray X1 provides limited capabilities for software prefetching with a scalar data prefetch instruction, it is unclear how compiler developers could exploit the feature due to the lack of inline assembly support. Stores are still non-blocking, and as mentioned in Section 5.1, we pipeline outstanding relaxed scalar puts and avoid the individual synchronization calls that block for their completion, taking advantage of hardware memory ordering on scalar conflicting accesses. Such code generation strategy is necessary to enhance the vectorizer's ability to optimize inner-loop fine-grained writes, but does not benefit programs with coarse-grained parallelism. Essentially, the X1's "vectorize or else" approach toward both communication and computation means that application and compiler developers have no direct control of an application's parallel performance, other than to apply transformations that result in the most vectorization. This heavy reliance on vectorization to achieve speedup is clearly not as flexible as split phase operations, and does not bode well for applications whose communication patterns are difficult to vectorize. For example, programs with distributed pointer-based data structures likely would not benefit from vectorization at all, while compiler-controlled data prefetching can be efficiently implemented with nonblocking remote accesses. Furthermore, a loop becomes less likely to vectorize as its length increases, meaning that complex applications that need communication optimizations the most will benefit less from vectorization; split-phase operations, on the contrary, operate on individual accesses and is thus less susceptible to this problem.

7 Parallel Performance

The NAS Multigrid (MG) benchmark was used to evaluate our compiler's parallel performance, as the program con-

[2]The MPI Fortran code only works for threads in powers of 2.

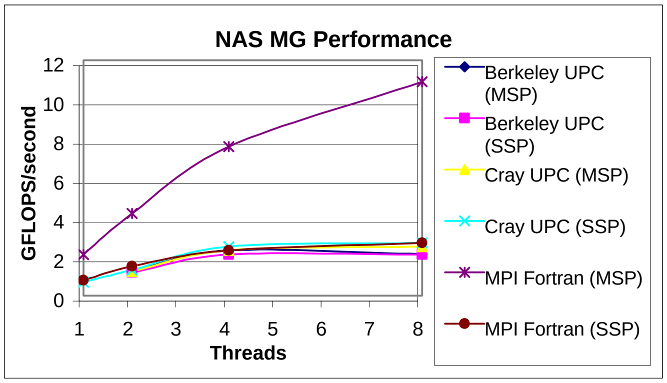


Figure 15. NAS MG (CLASS B).

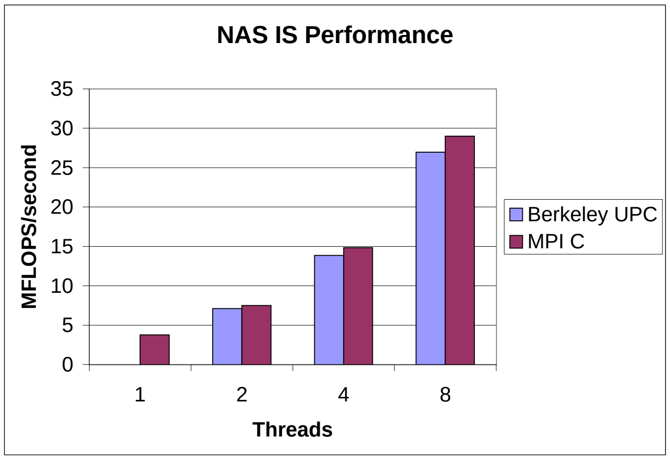


Figure 16. NAS IS (CLASS B).

tains a good balance of computation and communication. Running in both SSP and MSP mode, we compared three configurations: UPC compiled with Berkeley UPC, UPC with Cray UPC, and finally Fortran MPI with Cray Fortran. The Cray C compiler fails to automatically vectorize the computation loops in UPC code, and we had to explicitly insert the `#pragma concurrent` to declare the absences of recurrences when using either Berkeley or Cray UPC. As Figure 15 shows, all three compilers perform well in the absolute sense with performance in the giga-flops range, and Berkeley UPC somewhat slower compared to the other two compilers. The Fortran compiler, however, is able to take advantage of multi-streaming under MSP mode, outperforming the SSP mode by a ratio of about three, compared to the theoretic peak of four. Automatic multi-streaming unfortunately fails under the Cray C compiler, and we have found no reliable directives that force its application to loop nests. All three compilers exhibit an anomaly in scaling from 4 nodes to 8 nodes under SSP mode, where the spawner attempts to place all threads in the same node. An MSP now has more than one SSP executing in parallel, thus possibly causing cache interferences and memory contention. The results suggest that in SSP mode UPC can offer comparable performance as the MPI Fortran programming model if sufficient optimization hints are given to the Cray C compiler. The Cray C compiler's ability to automatically multi-stream a loop and effectively schedule the unused SSPs in MSP mode, however, appears limited. Another observation is that it seems more profitable to compile parallel applications in MSP mode; even if the compiler could not efficiently parallelize loops via multi-streaming, treating a node as a 16-way SMP and executing more than one thread in the same MSP will likely lead to contentions to the network and memory subsystem and thereby result in performance degradation.

In our performance study we next used the NAS integer sort (IS) kernel, a benchmark written in bulk synchronous style with high communication requirements. A UPC version of the benchmark compiled with Berkeley UPC (Cray UPC produces incorrect results) was compared against another version written with MPI in C. Both versions were compiled with full optimizations enabled, with all pointers declared restrict. Also, we do not distinguish between SSP and MSP mode, as the benchmark contains no loops that can profit from multi-streaming. As Figure 16 shows, Berkeley UPC achieves similar performance to MPI, with both scaling well for inter-node communications. In terms of absolute performance, however, both versions are quite inefficient, achieving significantly less than 1% peak performance on the X1. This is likely due to the fact that loops in the benchmark have recurrences and thus do not benefit from vectorization, but instead must be executed on the slower scalar processor. This supports our argument in Section 6.2 that questions the elimination of split-phase remote gets from the X1; whereas vectorization has failed to optimize the IS benchmark, split-phase operators could still be used to convert the remote bulk transfers into non-blocking operations and overlap the communication time with independent computation.

8 Analysis and Conclusions

In this paper, we have described and evaluated our implementation of the Berkeley UPC Compiler on the new Cray X1 architecture. Benchmarking results show that our compiler performs comparably to both the native Cray UPC Compiler and the MPI programming model on a vector NUMA platform. In particular, the GASNet layer offers communication performance that matches or exceeds the X1's *shmem* API, the translator sufficiently maintains the vectorizability of sequential code, and the runtime efficiently implements operations on pointer-to-shared. Our compiler also has the advantage that it fully supports the UPC 1.1 specification, while the omission of several important features in Cray UPC substantially reduces the language's versatility.

The architecture of Berkeley UPC has proved to be useful in evaluating the potential of the X1 for global address space languages. We have been able to tailor the implementation (and *not* the interface) of each component to a platform with characteristics that diverge substantially from existing tightly-coupled multiprocessors and the growing family of networks of workstations. On the X1, the layered architecture achieves equal or better performance by allowing the lower layers to be compiled away for simple word-size accesses and leveraging optimized implementations of more involved network primitives. For example, other implementations of UPC on shared memory machines have struggled with primitives involving remote procedure calls, a functionality easily surmounted using the GASNet Core.

We believe that the reliance on vectorization for efficient performance of fine-grained memory accesses is over-generalized. While it can be extremely effective for code idioms such strided accesses and scatter/gather operations, vectorizing more complex loops that tightly integrate computation with communication could be challenging. Moreover, it lacks the flexibility provided by split-phase communication primitives that offer developers more direct control over their application's communication performance.

Similarly, X1's heavy dependence on vectorization has some negative consequences for serial performance due to its limited scalar processing power. Applications whose main computation loops contain recurrences (e.g., NAS IS) can be inefficient as it does not benefit from the greater computation power and memory bandwidth offered by the vector pipelines. The C compiler's ability to automatically identify candidates for vectorization could also be further improved, as demonstrated by the Livermore loop results and the fact that we had to manually insert multiple pragma directives to achieve acceptable performance on our parallel benchmarks.

Additionally, although we share the view of the designers with regards to the usefulness of transparent global memory access, dropping the rich set of user-level communication primitives that could be achieved on the Cray T3E through E-registers definitely impairs general use of fine-grained operations. In that sense, our ability to efficiently utilize the entire system using general low-level primitives finds a mismatch in the X1's lack of per-operation completion guarantees and in the inaccessible set of communication instructions. While some of these effects could be mitigated if the C compiler allowed inline assembly, sufficiently expressive and general user-level communication primitives remain the dominating missing feature. We have shown that our compiler can match the performance of the native Cray UPC compiler but believe that much of its potential remains underutilized, mostly in areas such as efficiently scheduling communication and computation to hide network latencies, compensating for slower scalar operations and making better overall use of the system.

References

[1] The Berkeley UPC Compiler, 2002. http://upc.nersc.gov.

[2] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model for parallel computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[3] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *Intl. Symp. on Comp. Architecture*, 1995.

[4] K. Berlin, J. Huan, M. Jacob, et al. Evaluating the impact of programming language features on the performance of parallel applications on cluster architectures. In *16th International Workshop on Languages and Compilers for Parallel Processing (LCPC)*, 2003.

[5] D. Bonachea. GASNet specification. Technical Report CSD-02-1207, University of California, Berkeley, 2002.

[6] D. Bonachea and J. C. Duell. Problems with using MPI 1.1 and 2.0 as compilation targets. In *2nd Workshop on Hardware/Software Support for High Performance Scientific and Engineering Computing (SHPSEC-03)*, 2003.

[7] S. Chakrabarti, M. Gupta, and J. Choi. Global communication analysis and optimization. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 68–78, 1996.

[8] W. Chen, D. Bonachea, J. Duell, P. Husband, C. Iancu, and K. Yelick. A performance analysis of the Berkeley UPC Compiler. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*, 2003.

[9] Cray assembly language (CAL) for cray X1 systems reference manual. http://www.cray.com/craydoc/20/manuals/S-2314-50/html-S-2314-50/.

[10] Cray C/C++ reference manual. http://www.cray.com/craydoc/manuals/004-2179-003/html-004-2179-003/.

[11] Cray X1 system overview. http://www.cray.com/craydoc/20/manuals/S-2346-23/html-S-2346-23/S-2346-23-toc.html.

[12] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. Eicken, and K. Yelick. Parallel programming in Split-C. In *Supercomputing1993 (SC1993)*, 1993.

[13] T. Dunigan, M. Fahey, J. White, and P. Worley. Early evaluation of the cray x1. In *Supercomputing 2003 (SC2003)*, 2003.

[14] T. El-Ghazawi, W. Carlson, and J. Draper. *UPC specification*, 2001. http://www.gwu.edu/ upc/documentation.html.

[15] A. Krishnamurthy and K. Yelick. Analyses and optimizations for shared address space programs. *Jorunal of Parallel and Distributed Computing*, 1996.

[16] C. Luk and T. Mowry. Compiler-based prefetching for recursive data structures. In *Architectural Support for Programming Languages and Operating Systems*, pages 222–233, 1996.

[17] S. Lumetta and D. Culler. Managing concurrent access for shared memory active messages. In *In Proceedings of the International Parallel Processing Symposium*, pages 272–279, 1998.

[18] Man page collections: Shared memory access (SHMEM). http://www.cray.com/craydoc/20/manuals/S-2383-22/S-2383-22-manual.pdf.

[19] F. McMahon. The Livermore Fortran Kernels: A computer test of the numerical performance range. Technical report, Lawrence Livermore National Laboratory, 1986.

[20] R. Numwich and J. Reid. Co-Array Fortran for parallel programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

[21] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *Architectural Support for Programming Languages and Operating Systems*, pages 26–36, 1996.

[22] A. Wakatani. Effectiveness of Message Strip-Mining for Regular and Irregular Communication. In *PDCS*, Oct 94.

[23] K. Yelick et al. Titanium: a high performance java dialect. In *proceedings of ACM 1998 Workshop on Java for High-Performance Network Computing*, 1998.

[24] Y. Zhu and L. Hendren. Communication optimizations for parallel C programs. *Jorunal of Parallel and Distributed Computing*, 58(2):301–312, 1999.